CSE 401 – Compilers

MiniJava IL

Hal Perkins

Winter 2009

Compile-time Processing

- Decide representation of run-time data values
- Decide where data will be stored
  - registers
  - format of stack frames
  - global memory
  - format of in-memory data structures (e.g. records, arrays)
- Generate machine code to do basic operations
  - just like interpreting expression, except generate code that will evaluate it later
- Do optimizations across instructions if desired

Compilation Plan

- Translate ASTs into linear sequence of simple statements called intermediate code (IL or IR)
  - Source-language, target-language independent
- Translate IL into target code

- Intermediate code generation focuses on simple representations of source constructs
- Target code generation focuses on constraints of particular target machines
- Different front ends and back ends can share IL
- IL can be optimized independently of each

Our Plan

- Look first at MiniJava’s IL and code gen
  - Next phase of project
- Then look at runtime representation of code/data on target machine(s) (x86 mainly)
- Survey of MiniJava’s backend
- More general view of instruction selection/ register allocation/etc.
- Then code optimization and other topics as we have time

MiniJava's Intermediate Language

- Want intermediate language to have simple, explicit operations (humans don't write IL programs)
- Use simple declaration primitives
  - global functions, global variables
  - no classes, no implicit method lookup, no nesting
- Use simple data types
  - ints, doubles, explicit pointers, records, arrays
  - no booleans
  - no class types, no implicit class fields
  - arrays are naked sequences; no implicit length or bounds checks
- Use explicit gotos instead of control structures
- Make all implicit checks explicit (e.g. array bounds checks)
- Implement method lookup via explicit data structures and code

MiniJava's IL (1)

```
Program ::= {GlobalVarDecl} {FunDecl}
GlobalVarDecl ::= Type ID [= Value] ;
Type ::= int | double | *Type
             | Type [] | { {Type ID}/, } | fun
Value ::= Int | Double | &ID
              | [ {Value}/, ] | { {ID = Value}/, }
FunDecl ::= Type ID ( {Type ID}/,)
               { {VarDecl} {Stmt} }
VarDecl ::= Type ID ;
Stmt ::= Expr ; | LHSExpr = Expr ;
          | iffalse Expr goto Label ;
          | iftrue Expr goto Label ;
          | goto Label ; | label Label ;
          | throw new Exception( String ) ;
          | return Expr ;
```

MiniJava's IL (2)

```
Expr ::= LHSExpr    | Unop Expr
        | Expr Binop Expr
        | Callee ( {Expr}/, )
        | new Type [ [ Expr ] ]
        | Int    | Double    | & ID
LHSExpr ::= ID    | * Expr
        | Expr -> ID [ [ Expr ] ]
Unop ::= -.int   |  -.double   |  not   |  int2double
Binop ::= (+|-|*|/).(int|double)
        | (<|<=|>=|>|==|!=).(int|double)
        | <.unsigned
Callee ::= ID | ( * Expr )
        | String
```

MiniJava’s IL Classes (1 of 6)

```
ILProgram: {ILGlobalVarDecl} {ILFunDecl}
ILGlobalVarDecl: ILType String
   IlInitializedGlobalVarDecl: ILValue
```
ILType
```
   ILIntType
   ILDoubleType
   ILPtrType: ILType
   ILSequenceType: ILType
   ILRecordType: {ILType String}
   ILCodeType
```

MiniJava's IL Classes (2 of 6)

```
ILValue
  ILIntValue: int
  ILDoubleValue: double
  ILGlobalAddressValue: ILGlobalVar
  ILLabelAddressValue: ILLabel
  ILSequenceValue: {ILValue}
  ILRecordValue: {ILValue String}

ILFunDecl: ILType String {ILFormalVarDecl}
           {ILVarDecl} {ILStmt}
ILVarDecl: ILType String
  ILFormalVarDecl
```

MiniJava's IL Classes (3 of 6)

```
ILStmt
   ILExprStmt: ILExpr
     ILAssignStmt: ILAssignableExpr
   ILConditionalBranchStmt: ILExpr ILLabel
     ILConditionalBranchFalseStmt
     ILConditionalBranchTrueStmt
   ILGotoStmt: ILLabel
   ILLabelStmt: ILLabel
   ILThrowExceptionStmt: String
   ILReturnStmt: ILExpr

ILLabel: String

ILGlobalVar: String
```

MiniJava's IL Classes (4 of 6)

```
ILVar: ILVarDecl

ILExpr
   ILAssignableExpr
     ILVarExpr: ILVar
     ILPtrAccessExpr: ILExpr
     ILFieldAccessExpr: ILExpr ILType String
        ILSequenceFieldAccessExpr: ILExpr
ILUnopExpr: ILExpr
   IL{Int,Double}NegativeExpr,
     ILLOgicalNegateExpr, ILIntToDoubleExpr
```

MiniJava's IL Classes (5 of 6)

```
ILBinopExpr: ILExpr ILExpr
   IL{Int,Double}{Add,Sub,Mul,Div,
                  Equal,NotEqual,
                  LessThan,LessThanOrEqual
                  GreaterThanOrEqual,
                  GreaterThan}Expr,
   ILUnsignedLessThanExpr
ILAllocateExpr: ILType
   ILAllocateSequenceExpr: ILExpr
ILIntConstantExpr: int
ILDoubleConstantExpr: double
   ILGlobalAddressExpr: ILGlobalVar
```

MiniJava's IL Classes (6 of 6)

```
ILGlobalAddressExpr: ILGlobalVar
ILFunCallExpr: ILType {ILExpr}
    ILDirectFunCallExpr: String
    ILIndirectFunCallExpr: ILExpr
    ILRuntimeCallExpr: String
```

Intermediate Code Generation

- Choose representations for source-level data types
  - translate each **`ResolvedType`** into **`ILType`**(s)
- Recursively traverse ASTs (**`lower`** operation) creating corresponding **`IL pgm`** – parallels typechecking and evaluation traversals
  - **`Expr`** ASTs create **`ILExpr`** ASTs
  - **`Stmt`** ASTs create **`ILStmt`** ASTs
  - **`MethodDecl`** ASTs create **`ILFunDecl`** ASTs
  - **`ClassDecl`** ASTs create **`ILGlobalVarDecl`** ASTs
  - …

Data Type Representation (1)

- What IL type to use for each source type?
  - what operations are we going to need on them?
- **`int, boolean, double`**?

Data Type Representations (2)

- What IL type to use for each source type?
  - what operations are we going to need on them?

-
  ```
  class B {
     int i;
     D j;
  }
  ```
- Instance of **`Class B?`**

Inheritance

- How to lay out subclasses
  - Subclass inherits from superclass
  - Subclass can be assigned to a variable of superclass type implying subclass layout must "match" superclass layout

  -
    ```
    class B {
            int i;
            D j;
    }
    class C extends B {
            int x;
            F y;
    }
    ```
- instance of **`class C`**

Methods

- How to translate a method?
- Use a function
  - name is "mangled": name of class + name of method
  - make **`this`** an explicit argument
- Example

  -
    ```
    class B { ...
              int m(int i, double d) { ...
              body ... }
    }
    ```
- **`B`**'s method **`m`** translates to

```
int B_m(*{...B...} this, int i, double d)
{ ... translation of body ... }
```

Implementing Method Invocation

```
class B { ...
   int m(...) { ... }
   E n(...) { ... }
}
class C extends B { ...
    int m(...) { ... }  // override
    F p(...) { ... }
}
B b1=new(B)
C c2=new(C)
B b2=c2
b1.m(...)
b1.n(...)
c2.m(...)
c2.n(...)
c2.p(...)
b2.m(...)
b2.n(...)
```

Methods via Function Pointers in Instances

- Store code pointer for each new method in each instance
- Initialize with right method for that name for that object
- Do “instance var lookup” to get code pointer to invoke

```
class B { int i;
  int m(...) { ... }
  E n(...) { ... }
 }
class C extends B { int j;
  int m(...) { ... }  // override
  F p(...) { ... }
}
```

- Instance of **`class B`**:

```
*(int i, *code m, *code n}
```

- Instance of **`class C`**:

```
*(int i, *code m, *code n, int j, *code p}
```

Manipulating Method Function Ptrs

- Example

```
B b1 = new B();
C c2 = new C();
B b2 = c2;
b1.m(3, 4.5);
b2.m(3, 4.5);
```

- Translation:

```
*.. b1 = alloc {...B...}
b1->i = 0; b1->m = &B_m; b1->n = &B_n;
*.. c2 = alloc {...C...};
c2->i = 0; c2->m = &C m; c2->n = &B n;
c2->j = 0; c2->p = &C_p;
*.. b2 = c2
(*(b1->m)) (b1, 3, 4.5);
(*(b2->m)) (b2, 3, 4.5);
```

Shared Method Function Pointers

- All direct instances of a class store the same method function pointers
- So, can factor out common values into a single record shared by all -- often called a virtual function table, or vtbl
  - smaller objects, faster object creation
  - slower method invocations
- B's virtual function table (a global initialized variable):

```
{*code m, *code n} B_vtbl = {m=&B_m, n=&B_n};
```

- Example:

```
B b1 = new B();
b1.m(3, 4.5);
```

- Translation

```
*.. b1 = alloc{int i, *{...B_vtbl...} vtbl};
b1->i=0; b1->vtbl = &B_vtbl;
(*((b1->vtbl)->m))(b1, 3, 4.5);
```

Method Inheritance

- A subclass inherits all the methods of its superclasses: its method record includes all fields of its superclass
- Virtual function tables of subclass extends that of superclass with new methods, replacing overridden methods

```
 class B {int i;
   int m(...) { ... }
   E n(...) { ... }
  }
 class C extends B { int j;
   int m(...) { ... }  // override
   F p(...) { ... }
 }
{*code m, *code n} B_vtbl = {m=&B_m, n=&B_n};
{*code m, *code n, *code p} C_vtbl= {m=&C_m, n=&B_n,
                                     p=&C_p};
```

Example

- Example

```
B b1 = new B();
C c2 = new C();
B b2 = c2;
b1.m(3, 4.5);
b2.m(3, 4.5);
```

- Translation

```
*.. b1 = alloc {int i, *{...B_vtbl...} vtbl};
b1->i = 0; b1->vtbl = &B_vtbl;
*.. c2 = alloc {int i, *{...C_vtbl...} vtbl,int j};
c2->i = 0; c2->vtbl = &C_vtbl; c2->j = 0;
*.. b2 = c2
(*((b1->vtbl)->m)) (b1, 3, 4.5);
(*((b2->vtbl)->m)) (b2, 3, 4.5);
```

Main ICG Operations

`ILProgram Program.lower();`

- translate the whole program into an ILProgram

`void ClassDecl.lower(ILProgram);`

- translate method decls
- declare the class's method record (vtbl)

`void MethodDecl.lower(ILProgram, ClassSymbolTable);`

- translate into IL fun decl, add to IL program

`void Stmt.lower(ILFunDecl);`

- translate into IL statement(s), add to IL fun decl

`ILExpr Expr.evaluate(ILFunDecl);`

- translate into IL expr, return it

```
ILType Type.lower();
ILType ResolvedType.lower();
```

- return corresponding IL type

An Example ICG Operation

```
class IntLiteralExpr extends Expr {
   int value;
   ILExpr lower(ILFunDecl fun) {
      return new ILIntConstantExpr(value);
   }
}
```

An Example ICG Operation

```
class AddExpr extends Expr {
   Expr arg1;
   Expr arg2;
   ILExpr lower(ILFunDecl fun) {
      ILExpr arg1_expr = arg1.lower(fun);
      ILExpr arg2_expr = arg2.lower(fun);
      return new ILIntAddExpr(arg1_expr,
                              arg2_expr);
   }
}
```

Example Overloaded ICG Operation

```
class EqualExpr extends Expr {
  Expr arg1;
  Expr arg2;
  ILExpr lower(ILFunDecl fun) {
    ILExpr arg1_expr = arg1.lower(fun);
    ILExpr arg2_expr = arg2.lower(fun);
    if (arg1.getResultType().isIntType() &&
        arg2.getResultType().isIntType()) {
    return new ILIntEqualExpr(arg1_expr, arg2_expr);
    } else if (arg1.getResultType().isBoolType() &&
               arg2.getResultType().isBoolType()) {
      return new ILBoolEqualExpr(arg1_expr, arg2_expr);
    } else {
        throw new InternalCompilerError(...);
    }
  }
}
```

An Example ICG Operation

```
class VarDeclStmt extends Stmt {
    String name;
    Type type;
    void lower(ILFunDecl fun) {
       fun.declareLocal(type.lower(), name);
    }
}
```

`declareLocal` declares a new local variable in the IL function

ICG of Variable References

```
class VarExpr extends Expr {
   String name;
   VarInterface var_iface;  //set during typechecking
   ILExpr lower(ILFunDecl fun) {
       return var_iface.generateRead(fun);
   }
}
class AssignStmt extends Stmt {
   String lhs;
   Expr rhs;
   VarInterface lhs_iface;  //set during typechecking
   void lower(ILFunDecl fun) {
       ILExpr rhs_expr = rhs.lower(fun);
       lhs_iface.generateAssignment(rhs_expr, fun);
   }
}
```

`generateRead/generateAssignment` gen IL code to read/assign the variable
code depends on the kind of variable (local vs. instance)

ICG of Instance Variable Reference

```
class InstanceVarInterface extends VarInterface {
   ClassSymbolTable class_st;
   ILExpr generateRead(ILFunDecl fun) {
      ILExpr rcvr_expr =
         new ILVarExpr(fun.lookupVar("this"));
      ILType class_type =
         ILType.classILType(class_st);
      ILRecordMember var_member =
         class_type.getRecordMember(name);
      return new ILFieldAccessExpr(rcvr_expr,
                                   class_type,
                                   var_member);
}
```

ICG of Instance Variable Reference

```
void generateAssignment(ILExpr rhs_expr,
                        ILFunDecl fun) {
   ILExpr rcvr_expr =
      new ILVarExpr(fun.lookupVar("this"));
   ILType class_type =
      ILType.classILType(class_st);
   ILRecordMember var_member =
      class_type.getRecordMember(name);
   ILAssignableExpr lhs =
      new ILFieldAccessExpr(rcvr_expr,
                            class_type,
                            var_member);
   fun.addStmt(new ILAssignStmt(lhs, rhs_expr));
  }
}
```

ICG of `if` Statements

What IL code to generate for an if statement?

`if (`*testExpr*`)` *thenStmt* `else` *elseStmt*

if

```
class IfStmt extends Stmt {
  Expr test;
  Stmt then_stmt;
  Stmt else_stmt;
  void lower(ILFunDecl fun) {
    ILExpr test_expr = test.lower(fun);
    ILLabel false_label = fun.newLabel();
    fun.addStmt(
      new ILCondBranchFalseStmt(test_expr, false_label));
    then_stmt.lower(fun);
    ILLabel done label = fun.newLabel();
    fun.addStmt(new ILGotoStmt(done_label));
    fun.addStmt(new ILLabelStmt(false_label));
    else_stmt.lower(fun);
    fun.addStmt(new ILLabelStmt(done_label));
  }
}
```

ICG of Print Statements

- What IL code to generate for a print statement?
  - **`System.out.println(expr);`**
- No IL operations exist that do printing (or any kind of I/O)
  - oops

Runtime Libraries

- Can provide some functionality of compiled program in external runtime libraries
  - libraries written in any language, compiled separately
  - libraries can contain functions, data declarations
- Compiled code includes calls to functions & references to data declared libraries
- Final application links together compiled code and runtime libraries
- Often can implement functionality either through compiled code or through calls to library functions
  - tradeoffs?

ICG of Print Statements

```
class PrintlnStmt extends Stmt {
   Expr arg;
   void lower(ILFunDecl fun) {
     ILExpr arg_expr = arg.lower(fun);
     ILExpr call_expr =
        new ILRuntimeCallExpr("println_int",
                              arg_expr);
     fun.addStmt(new ILExprStmt(call_expr));
   }
}
```

```
// print out an integer value
void println_int(int value) {
  printf("%d\n", value);
}
```

- What about printing doubles?

ICG of new Expressions

- What IL code to generate for a new expression?

```
class C extends B {
    inst var decls
    method decls
}
... new C() ...
```

ICG of new Expressions

```
class NewExpr extends Expr {
   String class_name;
   ILExpr lower(ILFunDecl fun) {
      generate code to:
        allocate instance record
        initialize vtbl field with class's method record
        initialize inst vars to default values
      return reference to allocated record
   }
}
```

An Example ICG Operation

```
class MethodCallExpr extends Expr {
    String class_name;
    ILExpr lower(ILFunDecl fun) {
      generate code to:
        evaluate receiver and arg exprs
        test whether receiver is null
        load vtbl member of receiver
        load called method member of vtbl
        call fun ptr, passing receiver and args
      return call expr
    }
}
```

“allocate instance record”

```
// allocate bytes of memory, and return a pointer to it
void* allocate(int bytes) {
  return malloc(bytes);
}

// allocate base_bytes + num_elems*elems_bytes bytes of cleared
// memory, and return a pointer to it
void* allocate_sequence(int base_bytes, int num_elems, int elems_bytes)
   {
  return calloc(base_bytes + num_elems * elems_bytes, 1);
}

// report an exception and die
void throw_exception(char* message) {
  fprintf(stderr, "Unhandled exception: %s\n", message);
  exit(-1);
}
```

Where We Are

- So far we have a pretty low-level, but still abstract machine
- Next: x86 overview; mapping languages like MiniJava etc. to real processors
- Then:
  - MiniJava’s code generator
  - More elaborate backend algorithms
  - Fundamentals of compiler optimizations